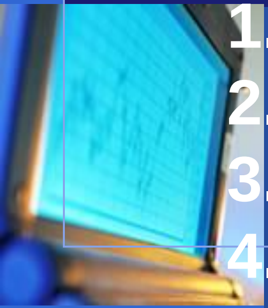
Brute Force



1. Sorting
2. Brute-Force string matching
3. Polynomial Evaluation
4. Closest pair problem by brute force

Brute Force

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved

Outline (Sorting)

- Several sorting algorithms:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- For each algorithm:
  - Basic Idea
  - Example
  - Implementation
  - Algorithm Analysis

Sorting

- Sorting = ordering.
- Sorted = ordered based on a particular way.
- Generally, collections of data are presented in a sorted manner.
- Examples of Sorting:
  - Words in a dictionary are sorted (and case distinctions are ignored).
  - Files in a directory are often listed in sorted order.
  - The index of a book is sorted (and case distinctions are ignored).
  - Many banks provide statements that list checks in increasing order (by check number).
  - In a newspaper, the calendar of events in a schedule is generally sorted by date.
  - Musical compact disks in a record store are generally sorted by recording artist.
- Why?
  - Imagine finding the phone number of your friend in your mobile phone, but the phone book is not sorted.

Bubble Sort: Idea

- Idea: bubble in water.
  - Bubble in water moves upward. Why?
- How?
  - When a bubble moves upward, the water from above will move downward to fill in the space left by the bubble.

Bubble Sort: Example

1	40	2	1	43	3	65	0	-1	58	3	42	4
2	2	1	40	3	43	0	-1	58	3	42	4	65
3	1	2	3	40	0	-1	43	3	42	4	58	65
4	1	2	3	0	-1	40	3	42	4	43	58	65

- Notice that at least one element will be in the correct position each iteration.

Bubble Sort: Example

5	1	2	0	-1	3	3	40	4	42	43	58	65
6	1	0	-1	2	3	3	4	40	42	43	58	65
7	0	-1	1	2	3	3	4	40	42	43	58	65
8	-1	0	1	2	3	3	4	40	42	43	58	65

Bubble Sort: Implementation

```
void sort(int a[]){
    for (int i = a.length; i>=0; i--) {
        boolean swapped = false;
        for (int j = 0; j<i; j++) {
            . . .
            if (a[j] > a[j+1]) {
                int T = a[j];
                a[j] = a[j+1];
                a[j+1] = T;
                swapped = true;
            }
            . . .
        }
        if (!swapped)
            return;
    }
}
```

Bubble Sort: Analysis

- Running time:
  - Worst case: $O(N^2)$

-

Selection Sort: Idea

1. We have two group of items:
   - sorted group, and
   - unsorted group
2. Initially, all items are in the unsorted group. The sorted group is empty.
   - We assume that items in the unsorted group unsorted.
   - We have to keep items in the sorted group sorted.
3. Select the "best" (eg. smallest) item from the unsorted group, then put the "best" item at the end of the sorted group.
4. Repeat the process until the unsorted group becomes empty.

Selection Sort: Example

40	2	1	43	3	65	0	-1	58	3	42	4

40	2	1	43	3	4	0	-1	58	3	42	65

40	2	1	43	3	4	0	-1	42	3	58	65

40	2	1	3	3	4	0	-1	42	43	58	65

Selection Sort: Example

40	2	1	3	3	4	0	-1	42	43	58	65

-1	2	1	3	3	4	0	40	42	43	58	65

-1	2	1	3	3	0	4	40	42	43	58	65

-1	2	1	0	3	3	4	40	42	43	58	65

Selection Sort: Example

-1	2	1	0	3	3	4	40	42	43	58	65
-1	0	1	2	3	3	4	40	42	43	58	65
-1	0	1	2	3	3	4	40	42	43	58	65
-1	0	1	2	3	3	4	40	42	43	58	65
-1	0	1	2	3	3	4	40	42	43	58	65

Selection Sort: Implementation

```
void sort(int a[]) throws Exception
{
    for (int i = 0; i < a.length; i++) {
        int min = i;
        int j;
        /*
          Find the smallest element in
          the unsorted list
        */
        for (j = i + 1; j < a.length; j++)
            . . .
            }
            if (a[j] < a[min]) {
                min = j;
            }
            . . .
        }
```

Selection Sort: Implementation

```
        /*
          Swap the smallest unsorted element
          into the end of the
          sorted list.
        */
        int T = a[min];
        a[min] = a[i];
        a[i] = T;
        ...
    }
}
```

Selection Sort: Analysis

- Running time:
  - Worst case: $O(N^2)$
- Based on big-oh analysis, is selection sort better than bubble sort?

Insertion Sort: Idea

- Idea: sorting cards.
  - 8 | 5 9 2 6 3
  - 5 8 | 9 2 6 3
  - 5 8 9 | 2 6 3
  - 2 5 8 9 | 6 3
  - 2 5 6 8 9 | 3
  - 2 3 5 6 8 9 |

Insertion Sort: Idea

1. We have two group of items:
   - sorted group, and
   - unsorted group
2. Initially, all items in the unsorted group and the sorted group is empty.
   - We assume that items in the unsorted group unsorted.
   - We have to keep items in the sorted group sorted.
3. Pick any item from, then insert the item at the right position in the sorted group to maintain sorted property.
4. Repeat the process until the unsorted group becomes empty.

Insertion Sort: Example

40	2	1	43	3	65	0	-1	58	3	42	4

2	40	1	43	3	65	0	-1	58	3	42	4

1	2	40	43	3	65	0	-1	58	3	42	4

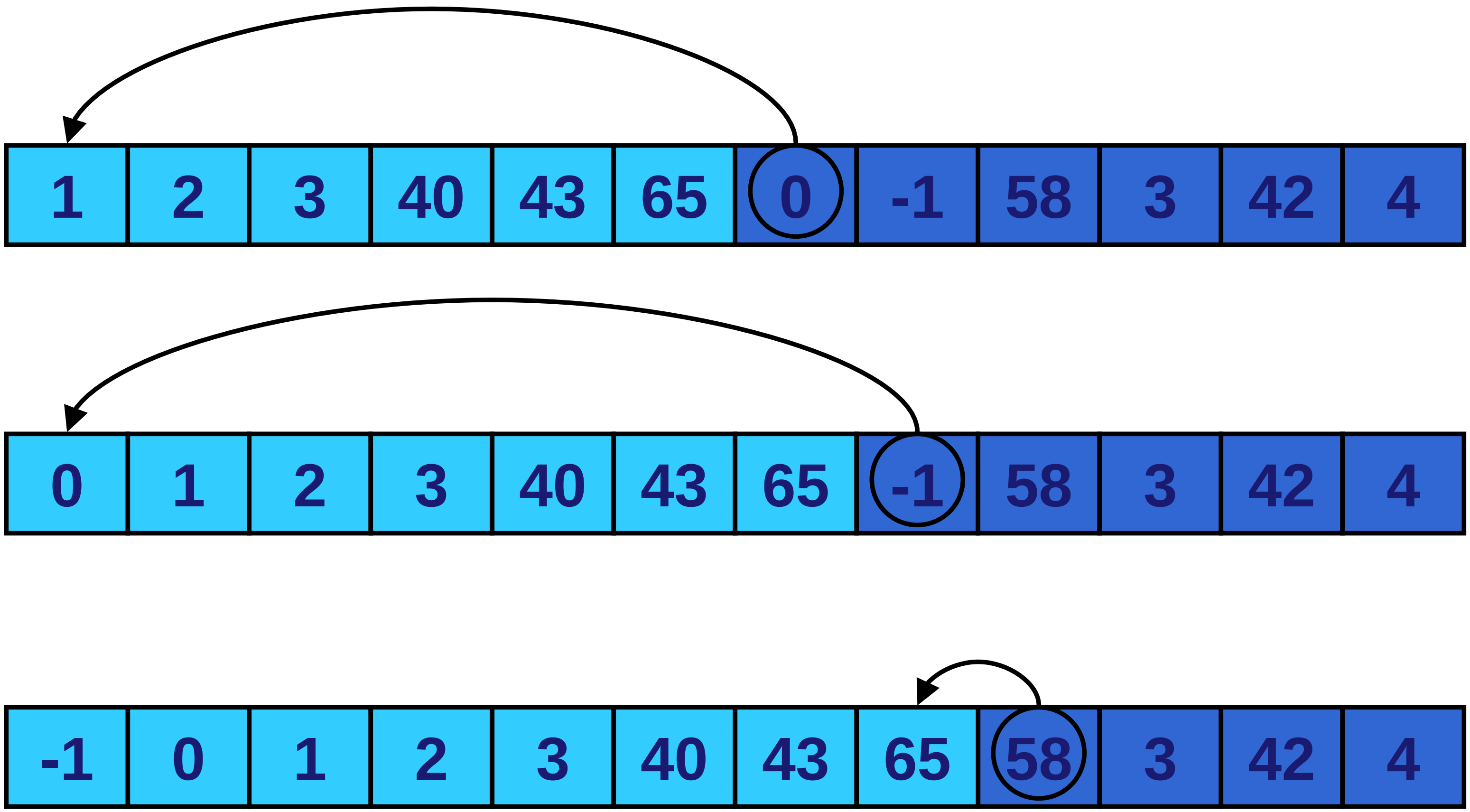
Insertion Sort: Example

1	2	40	43	3	65	0	-1	58	3	42	4

1	2	3	40	43	65	0	-1	58	3	42	4

1	2	3	40	43	65	0	-1	58	3	42	4

Insertion Sort: Example



Insertion Sort: Example

-1	0	1	2	3	40	43	58	65	3	42	4

-1	0	1	2	3	3	40	43	58	65	42	4

-1	0	1	2	3	3	40	42	43	58	65	4

-1	0	1	2	3	3	4	40	42	43	58	65

Sorting: Insertion Sort

```
InsertionSort(A, n) {
  for i = 2 to n {
     key = A[i]
     j = i - 1;
     while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
     }
     A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	10	40	20
1	2	3	4

$i = \varnothing$ $j = \varnothing$ $key = \varnothing$

$A[j] = \varnothing$ $A[j+1] = \varnothing$

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	10	40	20
1	2	3	4

i = 2   j = 1   key = 10
A[j] = 30   A[j+1] = 10

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2   j = 1   key = 10
A[j] = 30   A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2$ $j = 1$ $key = 10$

$A[j] = 30$ $A[j+1] = 30$

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 30$

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

30	30	40	20
1	2	3	4

i = 2   j = 0   key = 10
A[j] = ∅   A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 2$ $j = 0$ $key = 10$

$A[j] = \varnothing$ $A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 10$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3$ $j = 0$ $key = 40$

$A[j] = \varnothing$ $A[j+1] = 10$

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3 \quad j = 0 \quad key = 40$

$A[j] = \varnothing \quad A[j+1] = 10$

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 2 key = 40
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 2 key = 40
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 3 j = 2 key = 40
A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 4$ $j = 2$ $key = 40$

$A[j] = 30$ $A[j+1] = 40$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 2 key = 20

A[j] = 30 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4   j = 2   key = 20

A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 4$ $j = 3$ $key = 20$

$A[j] = 40$ $A[j+1] = 20$

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4 j = 3 key = 20
A[j] = 40 A[j+1] = 40

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4   j = 3   key = 20

A[j] = 40   A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

i = 4   j = 2   key = 20
A[j] = 30   A[j+1] = 40

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	40	40
1	2	3	4

$i = 4$ $j = 2$ $key = 20$
$A[j] = 30$ $A[j+1] = 40$

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
          A[j+1] = A[j]
          j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4 j = 2 key = 20
A[j] = 30 A[j+1] = 30

```
InsertionSort(A, n) {
  for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
            A[j+1] = A[j]
            j = j - 1
      }
      A[j+1] = key
  }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4   j = 1   key = 20

A[j] = 10   A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
      key = A[i]
      j = i - 1;
      while (j > 0) and (A[j] > key) {
         A[j+1] = A[j]
         j = j - 1
      }
      A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	30	30	40
1	2	3	4

i = 4   j = 1   key = 20
A[j] = 10   A[j+1] = 30

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	20	30	40
1	2	3	4

i = 4 j = 1 key = 20
A[j] = 10 A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

An Example: Insertion Sort

10	20	30	40
1	2	3	4

i = 4 j = 1 key = 20
A[j] = 10 A[j+1] = 20

```
InsertionSort(A, n) {
   for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
              A[j+1] = A[j]
              j = j - 1
        }
        A[j+1] = key
   }
}
```

Done!

Insertion Sort

InsertionSort(*A*, *n*)

for *i = 2* to *n* do

key = A[i]

j = i - 1 ———— c_1

while *j > 0* and *A[j] > key* do

A[j+1] = A[j]

j = j - 1 ———— c_2

A[j+1] = key

$$T(n) \le \sum_{i=2}^{n} (c_1 + c_2 i) = c_3 n^2 + c_4 n + c_5$$

Analysis

- Simplifications
  - Ignore actual and abstract statement costs
  - *Order of growth* is the interesting measure:
    - Highest-order term is what counts
      - Remember, we are doing asymptotic analysis
      - As the input size grows larger it is the high order term that dominates

Insertion Sort: Analysis

- Running time analysis:
  - Worst case: $O(N^2)$

- Is insertion sort faster than selection sort?
- Notice the similarity and the difference between insertion sort and selection sort.

A Lower Bound

- Bubble Sort, Selection Sort, Insertion Sort all have worst case of $O(N^2)$.
- Turns out, for any algorithm that exchanges adjacent items, this is the best worst case: $\Omega(N^2)$
- In other words, this is a lower bound!

Brute-Force String Matching

- *pattern*: a string of *m* characters to search for
- *text*: a (longer) string of *n* characters to search in
- problem: find a substring in the text that matches the pattern

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

Pseudocode and Efficiency

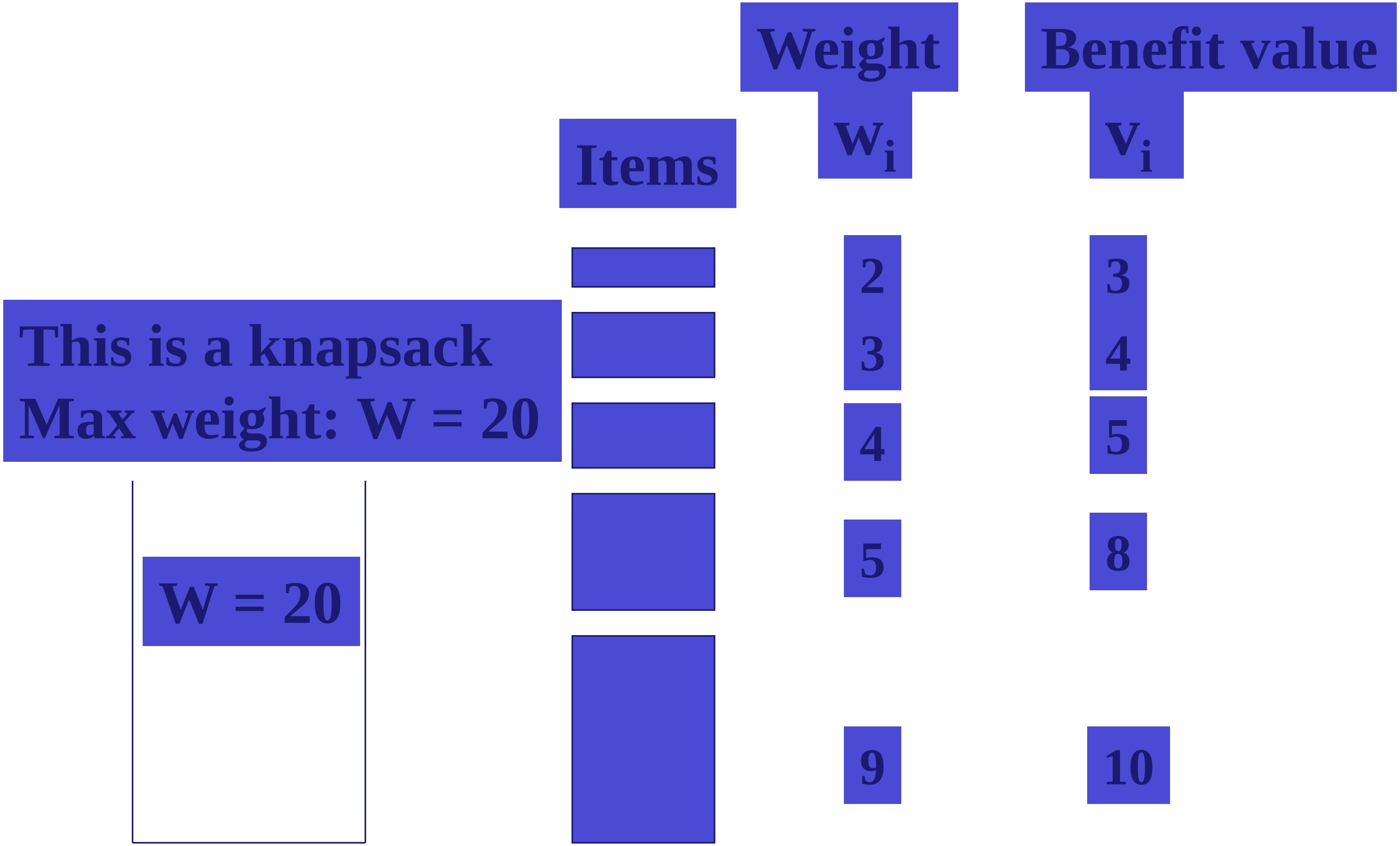
ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

//Implements brute-force string matching
//Input: An array $T[0..n-1]$ of n characters representing a text and
//   an array $P[0..m-1]$ of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//   matching substring or -1 if the search is unsuccessful
for $i \leftarrow 0$ **to** $n-m$ **do**
    $j \leftarrow 0$
    while $j < m$ **and** $P[j] = T[i+j]$ **do**
        $j \leftarrow j+1$
    if $j = m$ **return** i
return -1

0-1 Knapsack problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value v_i
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture



0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} v_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a "*0-1*" problem, because each item must be entirely accepted or rejected.
- In the "*Fractional Knapsack Problem*," we can take fractions of items.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- We go through all combinations (subsets) and find the one with maximum value and with total weight less or equal to *W*

Example 2: Knapsack Problem

Given *n* items:

- weights: $w_1 \quad w_2 \; \ldots \; w_n$
- values: $v_1 \quad v_2 \; \ldots \; v_n$
- a knapsack of capacity W

Find most valuable subset of the items that fit into the knapsack

Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

0-1 Knapsack problem: brute-force approach

- Algorithm:
  - We go through all combinations and find the one with maximum value and with total weight less or equal to *W*
- Efficiency:
  - Since there are n items, there are 2^n possible combinations of items.
  - Thus, the running time will be $O(2^n)$

Matrix-chain multiplication (MCM) - DP

- Problem: given <A_1, A_2, …,A_n>, compute the product: $A_1 \times A_2 \times \ldots \times A_n$, find the fastest way (i.e., minimum number of multiplications) to compute it.

- Suppose two matrices A(p,q) and B(q,r), compute their product C(p,r) in $p \times q \times r$ multiplications

```
for i=1 to p for j=1 to r C[i,j]=0
for i=1 to p
    for j=1 to r
        for k=1 to q
                C[i,j] = C[i,j]+ A[i,k]B[k,j]
```

Matrix-chain multiplication -DP

- Different parenthesizations will have different number of multiplications for product of multiple matrices

- Example: A(10,100), B(100,5), C(5,50)
  - If $((A \times B) \times C)$, $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$
  - If $(A \times (B \times C))$, $10 \times 100 \times 50 + 100 \times 5 \times 50 = 75000$

- The first way is ten times faster than the second !!!

- Denote $<A_1, A_2, \ldots, A_n>$ by $< p_0, p_1, p_2, \ldots, p_n>$
  - i.e, $A_1(p_0,p_1)$, $A_2(p_1,p_2)$, …, $A_i(p_{i-1},p_i)$,… $A_n(p_{n-1},p_n)$

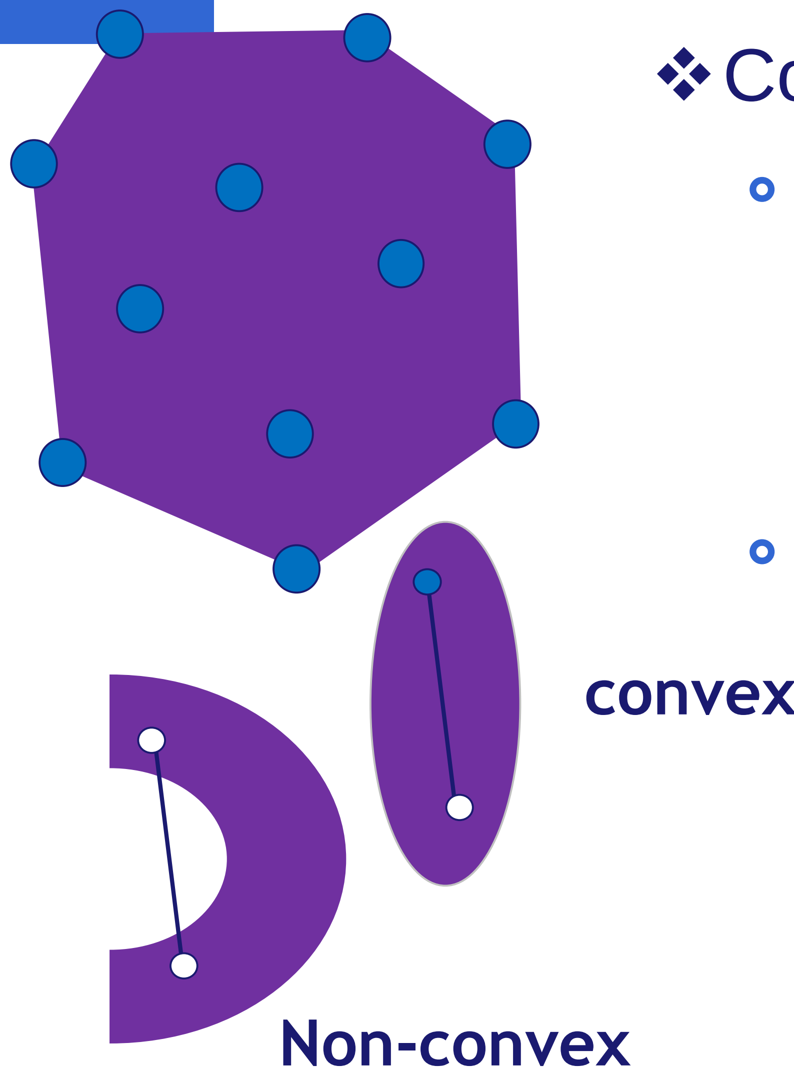
Matrix-chain multiplication –MCM DP

- Intuitive brute-force solution: Counting the number of parenthesizations by exhaustively checking all possible parenthesizations.

- Let P(n) denote the number of alternative parenthesizations of a sequence of n matrices:
  - $P(n) = \begin{cases} 1 & \text{if } n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$

- The solution to the recursion is $\Omega(2^n)$.
- So brute-force will not work.

Convex hull problem



- Convex hull
  - Problem:

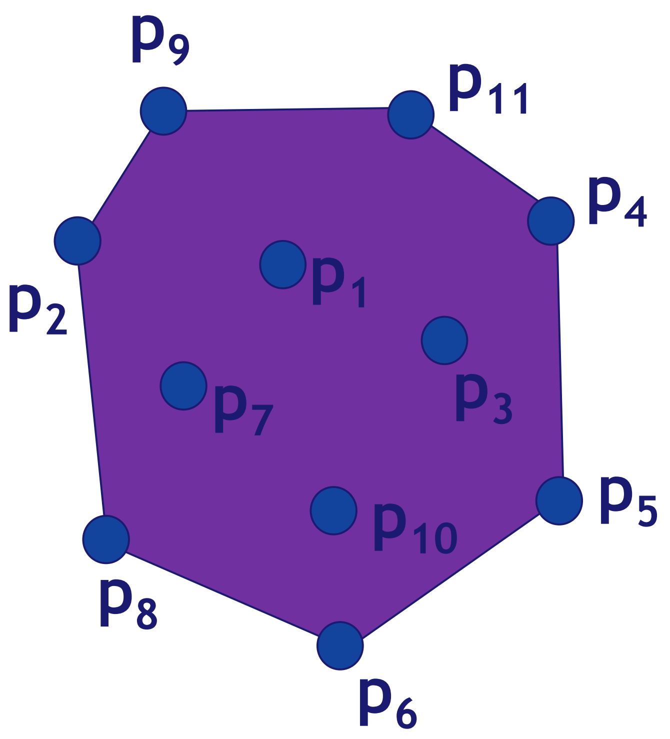
    Find <u>smallest convex polygon</u> enclosing *n* points on the plane

  - Convex:
    - A <u>geometric figure</u> with no indentations.
    - Formally, a geometric figure is convex if every <u>line segment</u> connecting <u>interior</u> <u>points</u> is entirely contained within the figure's interior.

Example: Convex Hull

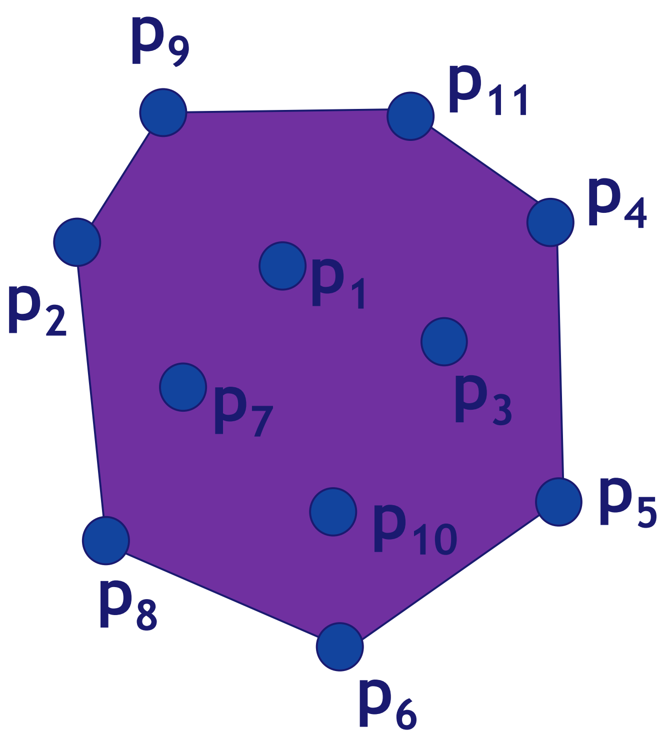
Input: $p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8,p_9,p_{10},p_{11}$

Output: $p_2,p_9,p_{11},p_4,p_5,p_6,p_8,p_2$



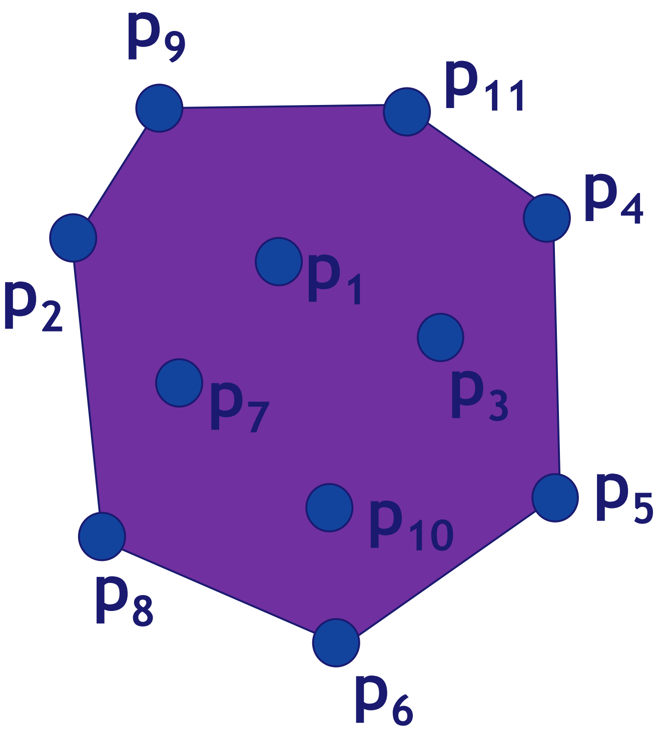
Convex hull brute force algorithm

- *Extreme points* of the convex polygon
  - Consider all the points in the polygon as a set. An *extreme point* is a point of the set that is not a middle point of any line segment with end points in the set.



Convex hull brute force algorithm

- *Extreme points* of the convex polygon
  - Consider all the points in the polygon as a set. An *extreme point* is a point of the set that is not a middle point of any line segment with end points in the set.



Which pairs of extreme points need to be connected to form the boundary of the convex hull?

Convex hull brute force algorithm

- A line segment connecting two points P_i and P_j of a set of n points is a part of its convex hull's boundary if and only if all the other points of the set lies on the same side of the straight line through these two points.

Convex hull brute force algorithm

- The straight line through two points $(x1, y1)$, $(x2, y2)$ in the coordinate plane can be defined by the following equation
  - $ax + by = c$

    where $a = y2 – y1$, $b = x1 – x2$, $c = x1y2 - y1x2$

- Such a line divides the plane into two half-planes: for all the points in one of them: $ax + by > c$, while for all the points in the other, $ax + by < c$.

Convex hull brute force algorithm

- Algorithm: For each pair of points p_1 and p_2 determine whether all other points lie to the same side of the straight line through p_1 and p_2, i.e. whether ax+by-c all have the same sign

- Efficiency:

Convex hull brute force algorithm

- Algorithm: For each pair of points p_1 and p_2 determine whether all other points lie to the same side of the straight line through p_1 and p_2, i.e. whether ax+by-c all have the same sign

- Efficiency: $\Theta(n^3)$

Brute force strengths and weaknesses

- Strengths:
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems
    - sorting; matrix multiplication; closest-pair; convex-hull
  - yields standard algorithms for simple computational tasks and graph traversal problems

Brute force strengths and weaknesses

- Weaknesses:
  - rarely yields efficient algorithms
  - some brute force algorithms unacceptably slow
    - e.g., the recursive algorithm for computing Fibonacci numbers
  - not as constructive/creative as some other design techniques